

Coping with NP-completeness: Exact Algorithms

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Advanced Algorithms and Complexity
Data Structures and Algorithms

Exact algorithms or intelligent exhaustive search: finding an optimal solution without going through all candidate solutions

Outline

- ① 3-Satisfiability
 - Backtracking
 - Local Search
- ② Traveling Salesman Problem
 - Dynamic Programming
 - Branch-and-bound

3-Satisfiability (3-SAT)

Input: A set of clauses, each containing at most three literals (that is, a 3-CNF formula).

Output: Find a satisfying assignment (if exists).

Examples

- The formula

$$(x \vee y \vee z)(x \vee \bar{y})(y \vee \bar{z})$$

is satisfiable: set $x = y = z = 1$ or
 $x = 1, y = z = 0$.

- The formula

$$(x \vee y \vee z)(x \vee \bar{y})(y \vee \bar{z})(z \vee \bar{x})(\bar{x} \vee \bar{y} \vee \bar{z})$$

is unsatisfiable.

A brute force search algorithm checking satisfiability of a 3-CNF formula F with n variables, goes through all assignments and has running time $O(|F| \cdot 2^n)$.

A brute force search algorithm checking satisfiability of a 3-CNF formula F with n variables, goes through all assignments and has running time $O(|F| \cdot 2^n)$.

Goal

Avoid going through all 2^n assignments

Outline

- 1 3-Satisfiability
 - Backtracking
 - Local Search
- 2 Traveling Salesman Problem
 - Dynamic Programming
 - Branch-and-bound

Main Idea of Backtracking

- Construct a solution piece by piece

Main Idea of Backtracking

- Construct a solution piece by piece
- Backtrack if the current partial solution cannot be extended to a valid solution

Example

$$(x_1 \vee x_2 \vee x_3 \vee x_4)(\bar{x}_1)(x_1 \vee x_2 \vee \bar{x}_3)(x_1 \vee \bar{x}_2)(x_2 \vee \bar{x}_4)$$

Example

$$(x_1 \vee x_2 \vee x_3 \vee x_4)(\bar{x}_1)(x_1 \vee x_2 \vee \bar{x}_3)(x_1 \vee \bar{x}_2)(x_2 \vee \bar{x}_4)$$

$$x_1 = 0$$

$$(x_2 \vee x_3 \vee x_4)(x_2 \vee \bar{x}_3)(\bar{x}_2)(x_2 \vee \bar{x}_4)$$

Example

$$(x_1 \vee x_2 \vee x_3 \vee x_4)(\bar{x}_1)(x_1 \vee x_2 \vee \bar{x}_3)(x_1 \vee \bar{x}_2)(x_2 \vee \bar{x}_4)$$

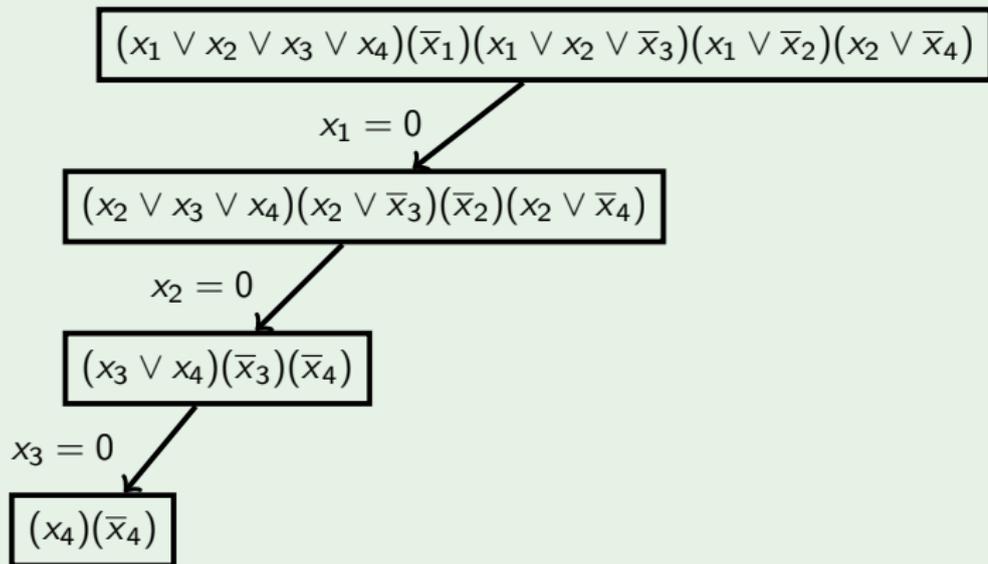
$$x_1 = 0$$

$$(x_2 \vee x_3 \vee x_4)(x_2 \vee \bar{x}_3)(\bar{x}_2)(x_2 \vee \bar{x}_4)$$

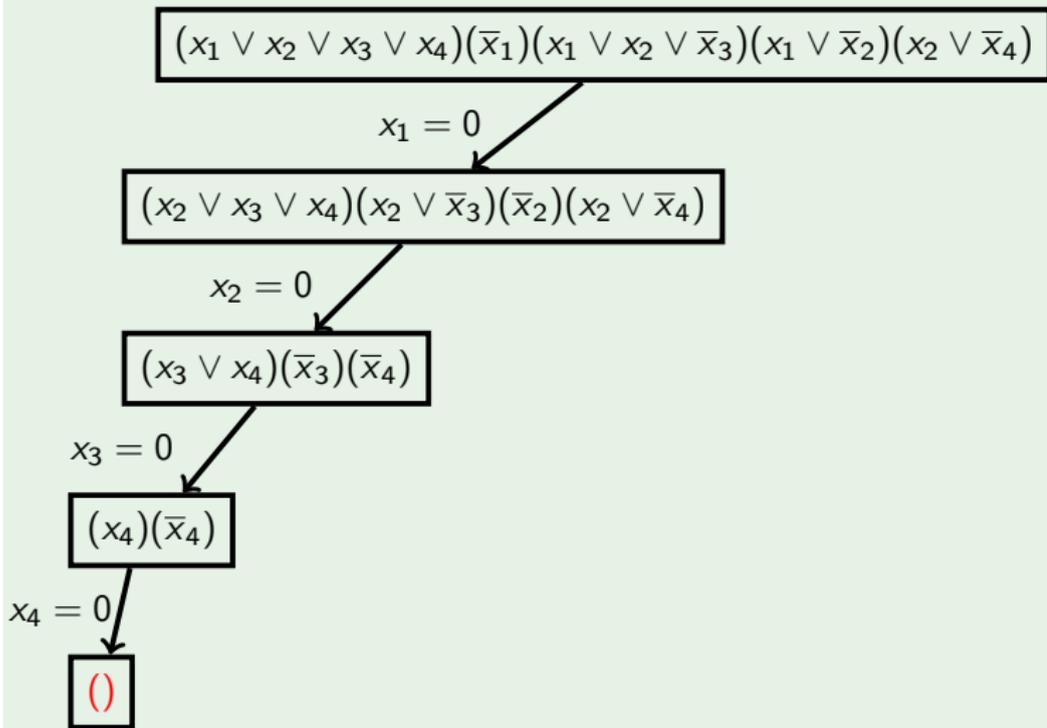
$$x_2 = 0$$

$$(x_3 \vee x_4)(\bar{x}_3)(\bar{x}_4)$$

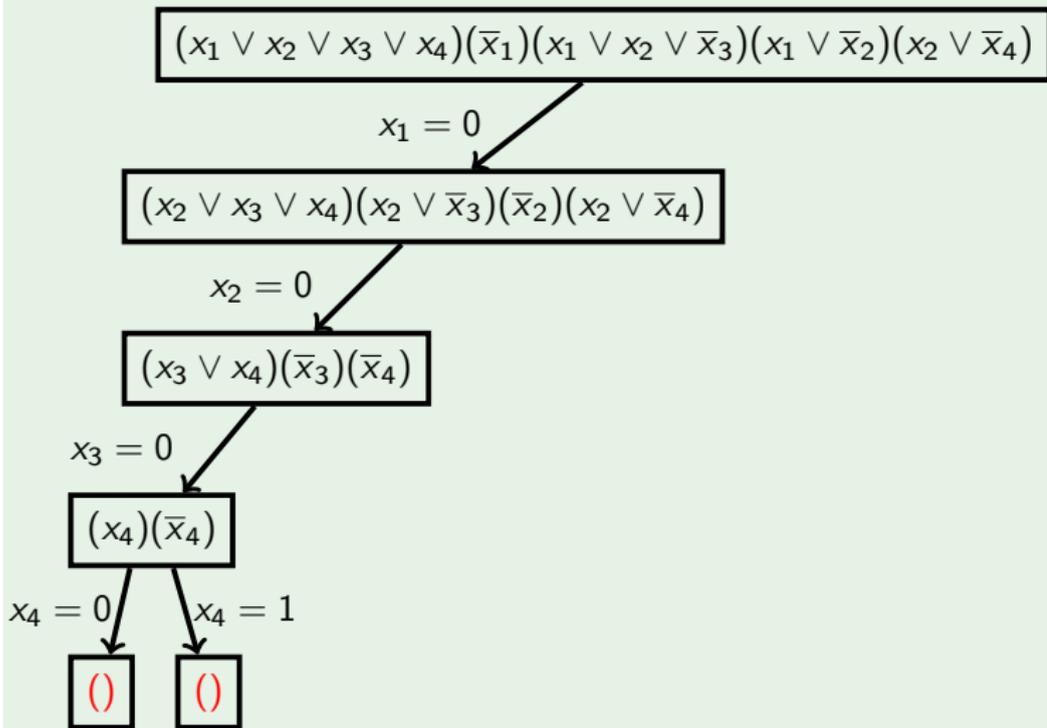
Example



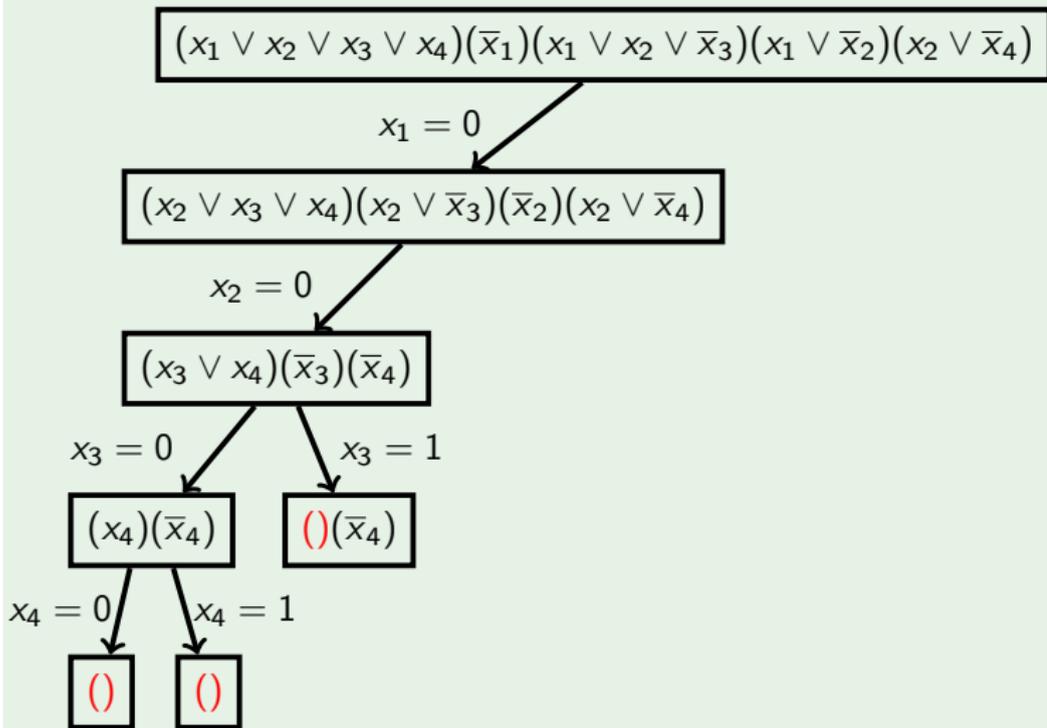
Example



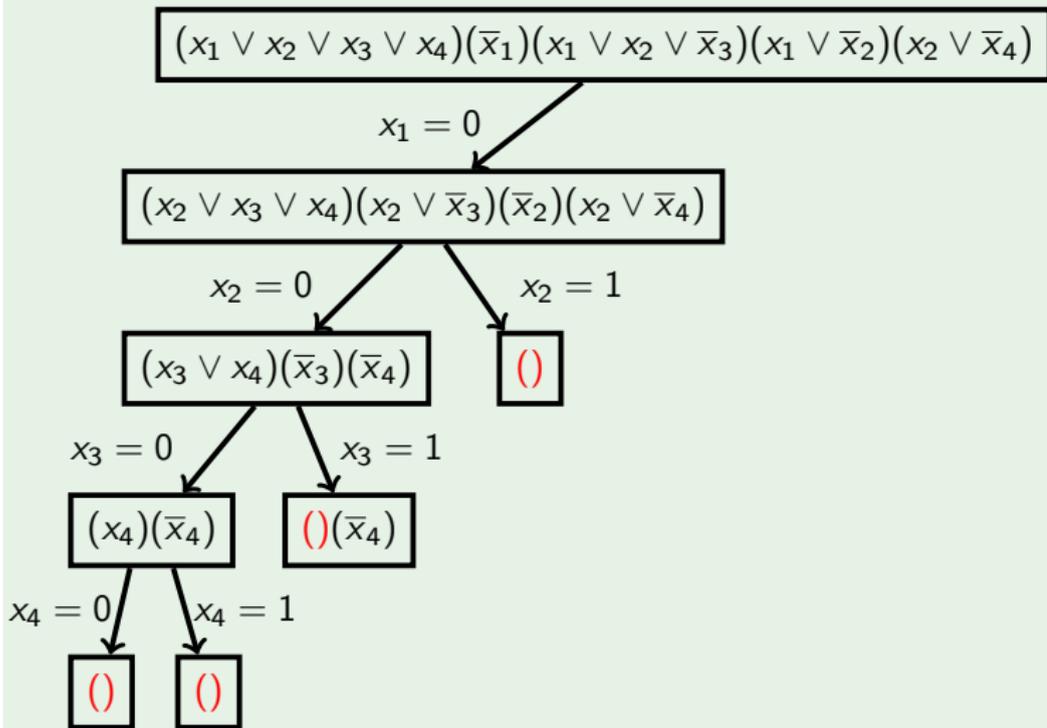
Example



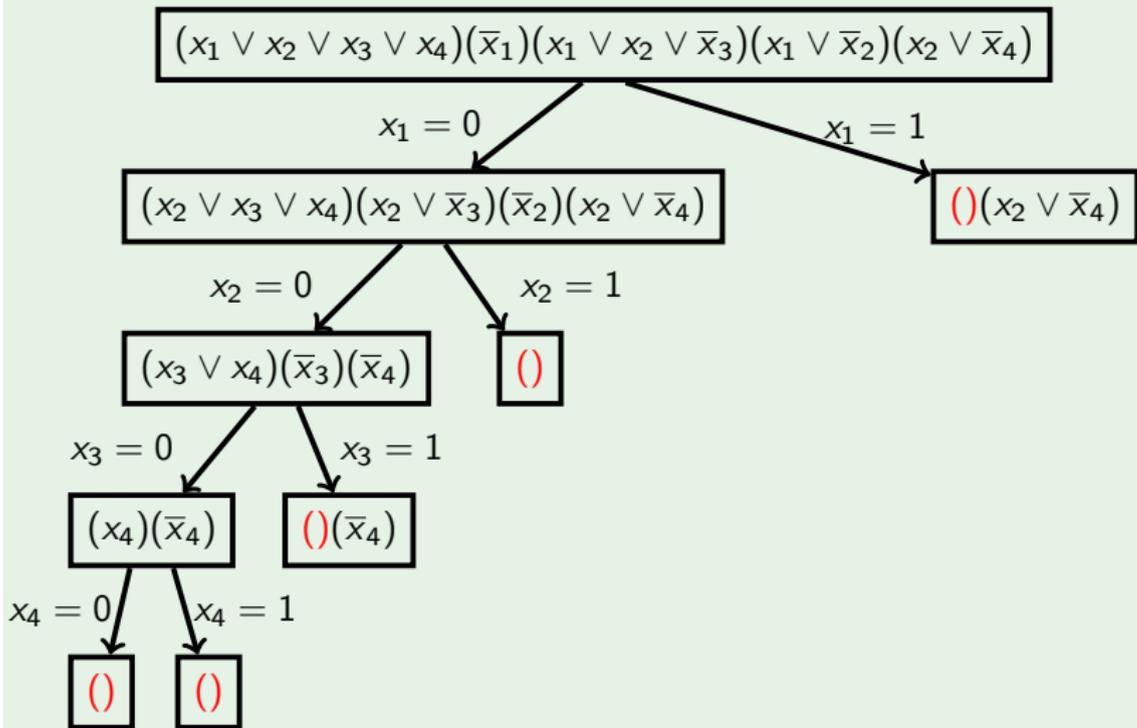
Example



Example



Example



SolveSAT(F)

```
if  $F$  has no clauses:  
    return “sat”
```

SolveSAT(F)

if F has no clauses:

 return “sat”

if F contains an empty clause:

 return “unsat”

SolveSAT(F)

if F has no clauses:

 return “sat”

if F contains an empty clause:

 return “unsat”

$x \leftarrow$ unassigned variable of F

SolveSAT(F)

if F has no clauses:

 return “sat”

if F contains an empty clause:

 return “unsat”

$x \leftarrow$ unassigned variable of F

if SolveSAT($F[x \leftarrow 0]$) = “sat”:

 return “sat”

SolveSAT(F)

if F has no clauses:

 return “sat”

if F contains an empty clause:

 return “unsat”

$x \leftarrow$ unassigned variable of F

if SolveSAT($F[x \leftarrow 0]$) = “sat”:

 return “sat”

if SolveSAT($F[x \leftarrow 1]$) = “sat”:

 return “sat”

SolveSAT(F)

if F has no clauses:

 return “sat”

if F contains an empty clause:

 return “unsat”

$x \leftarrow$ unassigned variable of F

if SolveSAT($F[x \leftarrow 0]$) = “sat”:

 return “sat”

if SolveSAT($F[x \leftarrow 1]$) = “sat”:

 return “sat”

return “unsat”

- Thus, instead of considering all 2^n branches of the recursion tree, we track carefully each branch

- Thus, instead of considering all 2^n branches of the recursion tree, we track carefully each branch
- When we realize that a branch is dead (cannot be extended to a solution), we immediately cut it

- Backtracking is used in many state-of-the-art SAT-solvers

- Backtracking is used in many state-of-the-art SAT-solvers
- SAT-solvers use tricky heuristics to choose a variable to branch on and to simplify a formula before branching

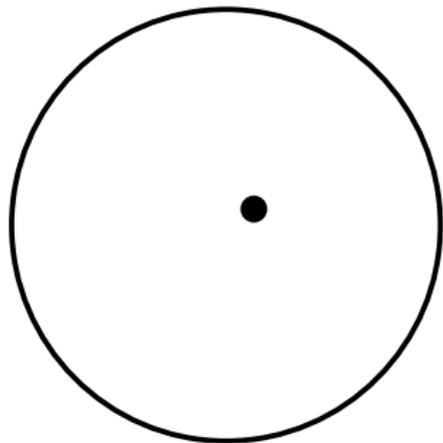
- Backtracking is used in many state-of-the-art SAT-solvers
- SAT-solvers use tricky heuristics to choose a variable to branch on and to simplify a formula before branching
- Another commonly used technique is local search — will consider it in the next part

Outline

- 1 3-Satisfiability
 - Backtracking
 - Local Search
- 2 Traveling Salesman Problem
 - Dynamic Programming
 - Branch-and-bound

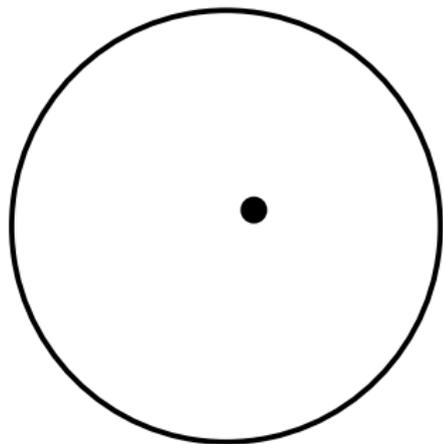
Main Idea of Local Search

- Start with a candidate solution



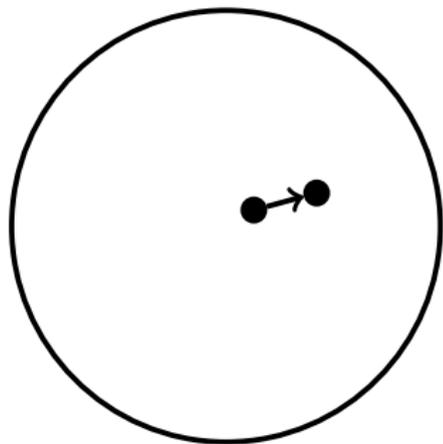
Main Idea of Local Search

- Start with a candidate solution
- Iteratively move from the current candidate to its neighbor trying to improve the candidate



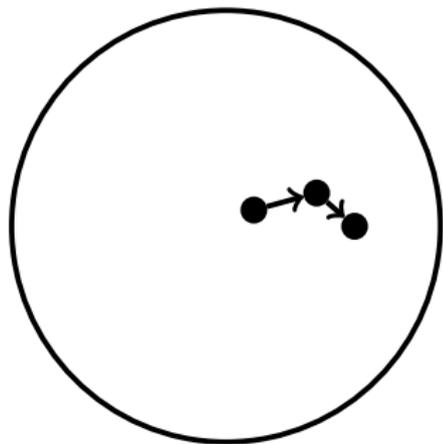
Main Idea of Local Search

- Start with a candidate solution
- Iteratively move from the current candidate to its neighbor trying to improve the candidate



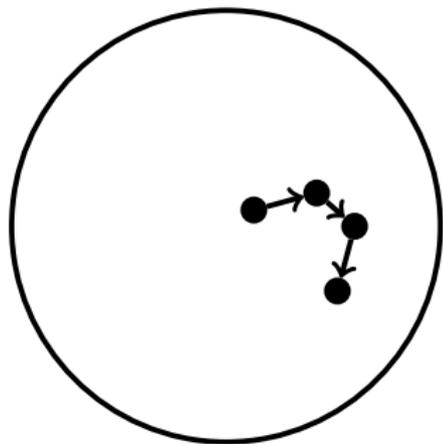
Main Idea of Local Search

- Start with a candidate solution
- Iteratively move from the current candidate to its neighbor trying to improve the candidate



Main Idea of Local Search

- Start with a candidate solution
- Iteratively move from the current candidate to its neighbor trying to improve the candidate



- Let F be a 3-CNF formula over variables x_1, x_2, \dots, x_n

- Let F be a 3-CNF formula over variables x_1, x_2, \dots, x_n
- A candidate solution is a truth assignment to these variables, that is, a vector from $\{0, 1\}^n$

Definition

Hamming distance (or just distance) between two assignments $\alpha, \beta \in \{0, 1\}^n$ is the number of bits where they differ:

$$\text{dist}(\alpha, \beta) = |\{i: \alpha_i \neq \beta_i\}|.$$

Definition

Hamming distance (or just distance) between two assignments $\alpha, \beta \in \{0, 1\}^n$ is the number of bits where they differ:

$$\text{dist}(\alpha, \beta) = |\{i: \alpha_i \neq \beta_i\}|.$$

Definition

Hamming ball with center $\alpha \in \{0, 1\}^n$ and radius r , denoted by $\mathcal{H}(\alpha, r)$, is the set of all truth assignments from $\{0, 1\}^n$ at distance at most r from α .

Example

- $\mathcal{H}(1011, 0) = \{1011\}$

Example

- $\mathcal{H}(1011, 0) = \{1011\}$
- $\mathcal{H}(1011, 1) =$
 $\{1011, 0011, 1111, 1001, 1010\}$

Example

- $\mathcal{H}(1011, 0) = \{1011\}$
- $\mathcal{H}(1011, 1) =$
 $\{1011, 0011, 1111, 1001, 1010\}$
- $\mathcal{H}(1011, 2) =$
 $\{1011, 0011, 1111, 1001, 1010,$
 $0111, 0001, 0010, 1101, 1110, 1000\}$

Searching a Ball for a Solution

Lemma

Assume that $\mathcal{H}(\alpha, r)$ contains a satisfying assignment β for F . We can then find a (possibly different) satisfying assignment in time $O(|F| \cdot 3^r)$.

Proof

- If α satisfies F , return α

Proof

- If α satisfies F , return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \bar{x}_j \vee x_k)$

Proof

- If α satisfies F , return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \bar{x}_j \vee x_k)$
- α assigns $x_i = 0, x_j = 1, x_k = 0$

Proof

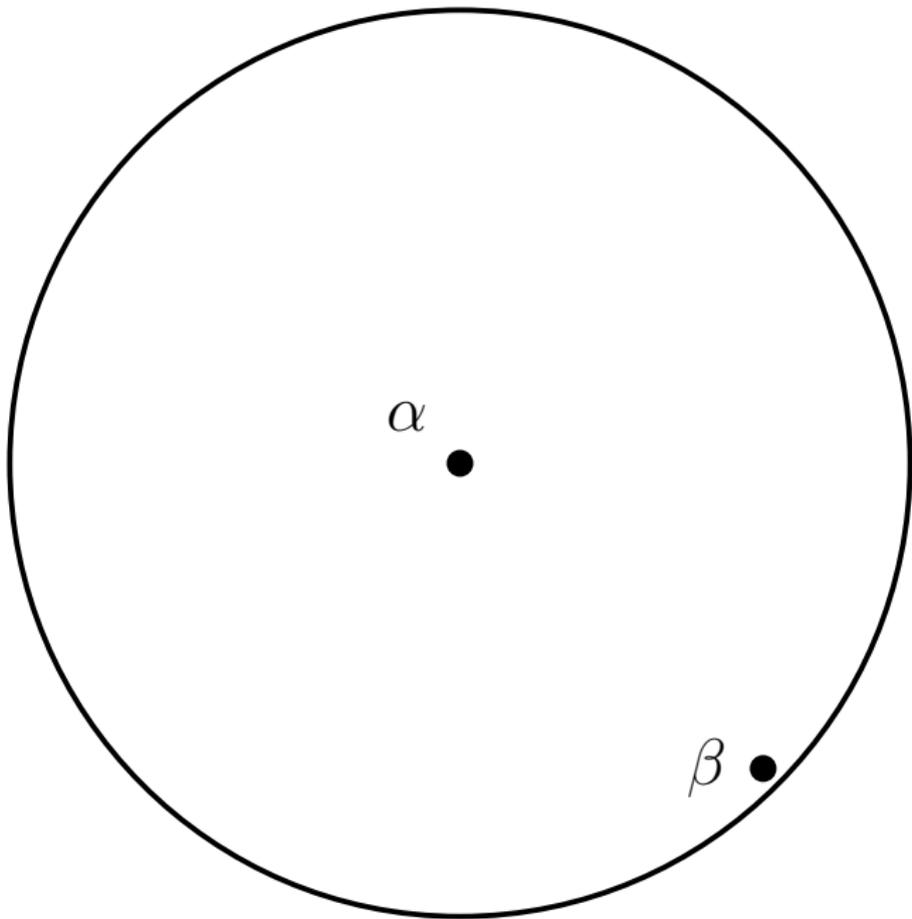
- If α satisfies F , return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \bar{x}_j \vee x_k)$
- α assigns $x_i = 0, x_j = 1, x_k = 0$
- Let $\alpha^i, \alpha^j, \alpha^k$ be assignments resulting from α by flipping the i -th, j -th, k -th bit, respectively

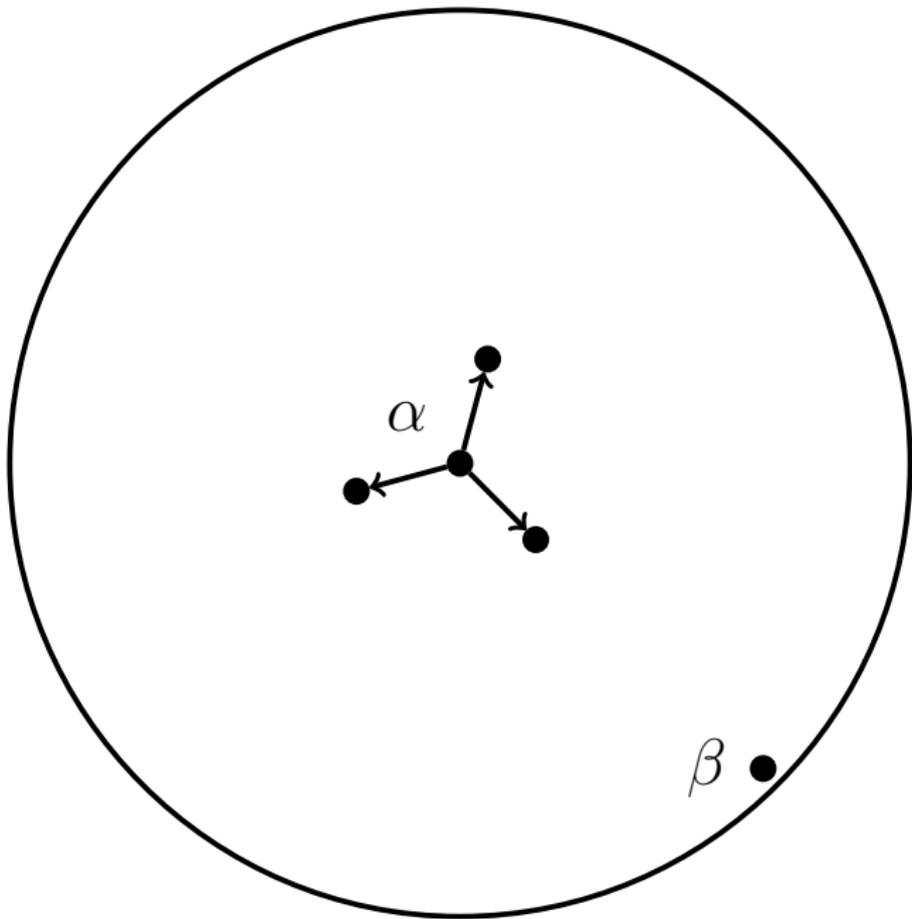
Proof

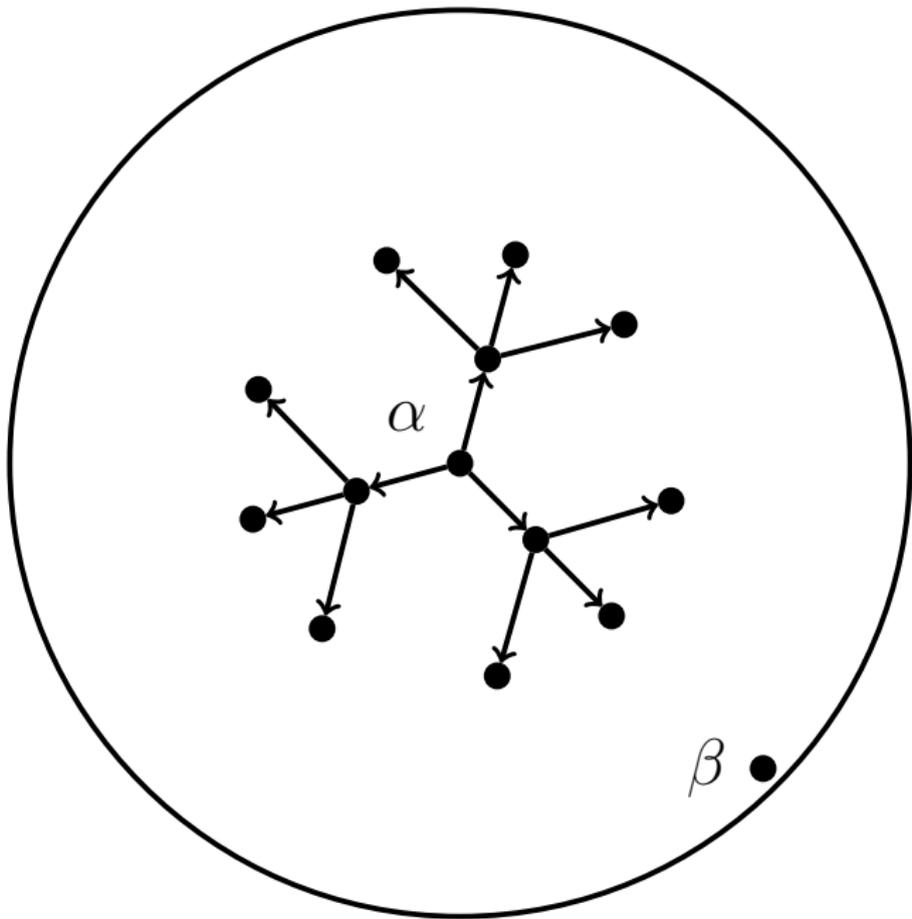
- If α satisfies F , return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \bar{x}_j \vee x_k)$
- α assigns $x_i = 0, x_j = 1, x_k = 0$
- Let $\alpha^i, \alpha^j, \alpha^k$ be assignments resulting from α by flipping the i -th, j -th, k -th bit, respectively
- **Crucial observation:** at least one of them is closer to β than α

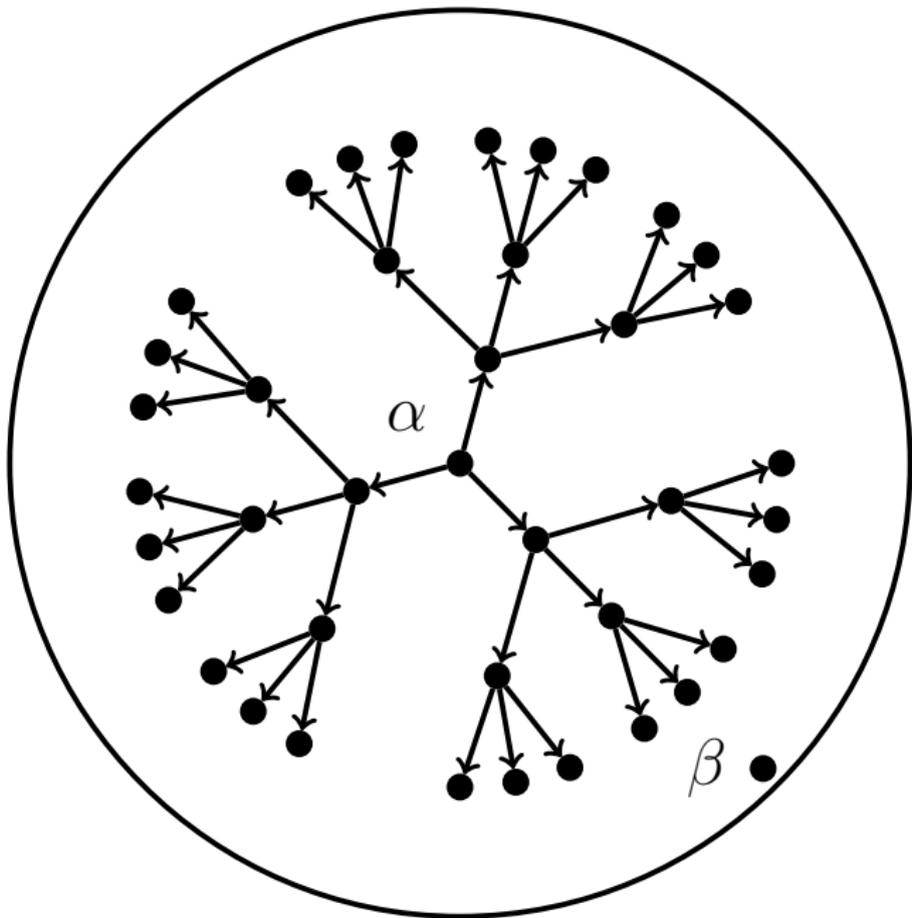
Proof

- If α satisfies F , return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \bar{x}_j \vee x_k)$
- α assigns $x_i = 0, x_j = 1, x_k = 0$
- Let $\alpha^i, \alpha^j, \alpha^k$ be assignments resulting from α by flipping the i -th, j -th, k -th bit, respectively
- **Crucial observation:** at least one of them is closer to β than α
- Hence there are at most 3^r recursive calls □









CheckBall(F, α, r)

```
if  $\alpha$  satisfies  $F$ :  
    return  $\alpha$ 
```

CheckBall(F, α, r)

```
if  $\alpha$  satisfies  $F$ :  
    return  $\alpha$   
if  $r = 0$ :  
    return "not found"
```

CheckBall(F, α, r)

if α satisfies F :

 return α

if $r = 0$:

 return “not found”

$x_i, x_j, x_k \leftarrow$ variables of unsatisfied clause

$\alpha^i, \alpha^j, \alpha^k \leftarrow \alpha$ with bits i, j, k flipped

CheckBall(F, α, r)

if α satisfies F :

 return α

if $r = 0$:

 return “not found”

$x_i, x_j, x_k \leftarrow$ variables of unsatisfied clause

$\alpha^i, \alpha^j, \alpha^k \leftarrow \alpha$ with bits i, j, k flipped

CheckBall($F, \alpha^i, r - 1$)

CheckBall($F, \alpha^j, r - 1$)

CheckBall($F, \alpha^k, r - 1$)

CheckBall(F, α, r)

if α satisfies F :

return α

if $r = 0$:

return “not found”

$x_i, x_j, x_k \leftarrow$ variables of unsatisfied clause

$\alpha^i, \alpha^j, \alpha^k \leftarrow \alpha$ with bits i, j, k flipped

CheckBall($F, \alpha^i, r - 1$)

CheckBall($F, \alpha^j, r - 1$)

CheckBall($F, \alpha^k, r - 1$)

if a satisfying assignment is found:

return it

else:

return “not found”

- Assume that F has a satisfying assignment β

- Assume that F has a satisfying assignment β
- If it has more 1's than 0's then it has distance at most $n/2$ from all-1's assignment

- Assume that F has a satisfying assignment β
- If it has more 1's than 0's then it has distance at most $n/2$ from all-1's assignment
- Otherwise it has distance at most $n/2$ from all-0's assignment

- Assume that F has a satisfying assignment β
- If it has more 1's than 0's then it has distance at most $n/2$ from all-1's assignment
- Otherwise it has distance at most $n/2$ from all-0's assignment
- Thus, it suffices to make two calls:
 $\text{CheckBall}(F, 11 \dots 1, n/2)$ and
 $\text{CheckBall}(F, 00 \dots 0, n/2)$

Running Time

- The running time of the resulting algorithm is

$$O(|F| \cdot 3^{n/2}) \approx O(|F| \cdot 1.733^n)$$

Running Time

- The running time of the resulting algorithm is

$$O(|F| \cdot 3^{n/2}) \approx O(|F| \cdot 1.733^n)$$

- On one hand, this is still exponential

Running Time

- The running time of the resulting algorithm is
 $O(|F| \cdot 3^{n/2}) \approx O(|F| \cdot 1.733^n)$
- On one hand, this is still exponential
- On the other hand, it is exponentially faster than a brute force search algorithm that goes through all 2^n truth assignments!

Outline

- 1 3-Satisfiability
 - Backtracking
 - Local Search
- 2 Traveling Salesman Problem
 - Dynamic Programming
 - Branch-and-bound

Traveling salesman problem (TSP)

Input: A complete graph with weights on edges and a budget b .

Output: A cycle that visits each vertex exactly once and has total weight at most b .

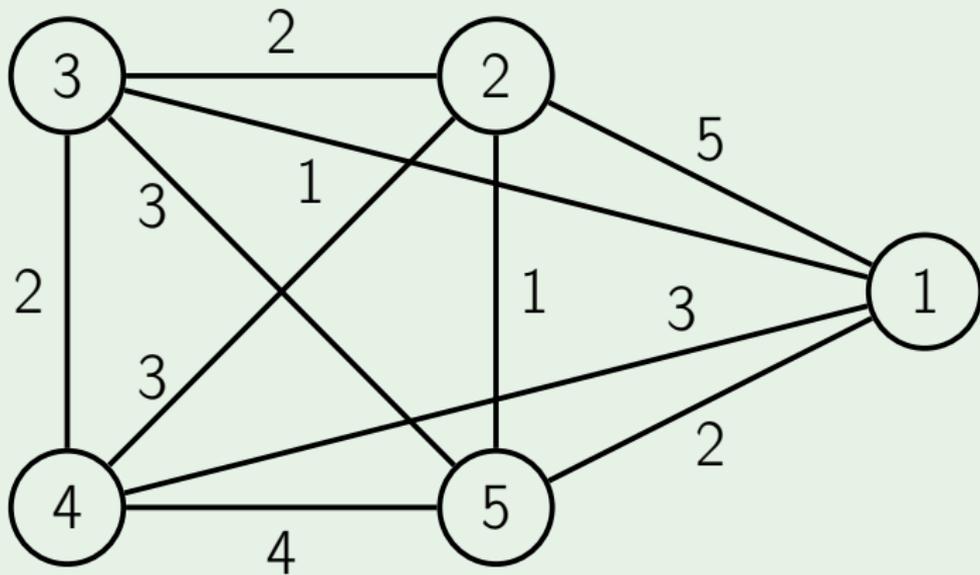
Traveling salesman problem (TSP)

Input: A complete graph with weights on edges and a budget b .

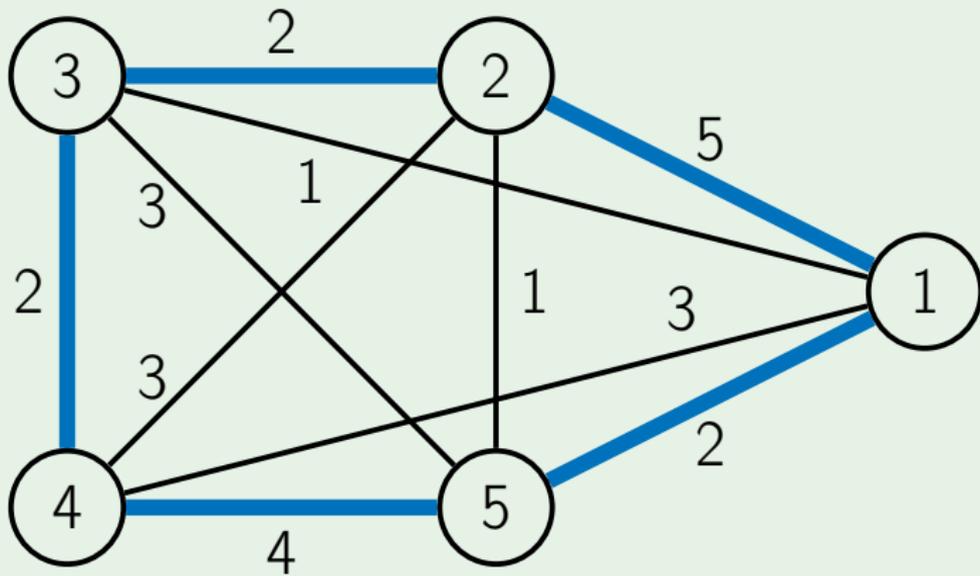
Output: A cycle that visits each vertex exactly once and has total weight at most b .

It will be convenient to assume that vertices are integers from 1 to n and that the salesman starts his trip in (and also returns back to) vertex 1.

Example

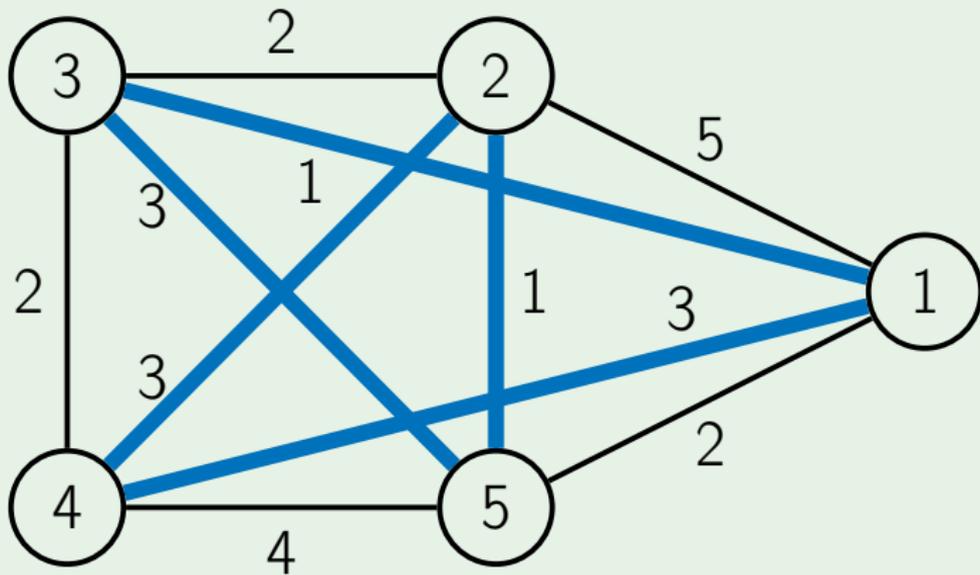


Example



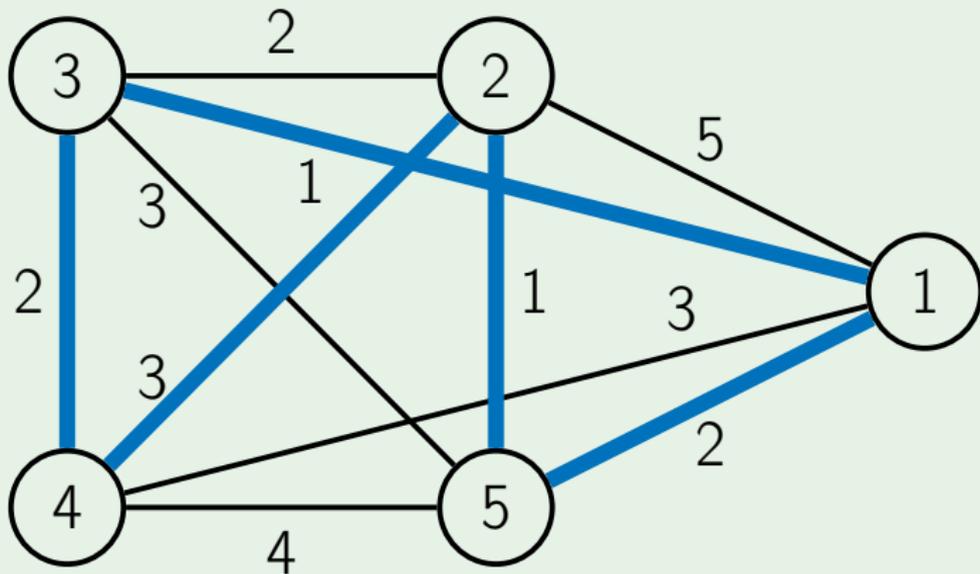
length: 15

Example



length: 11

Example



length: 9

Brute Force Solution

A naive algorithm just checks all possible $(n - 1)!$ cycles.

Brute Force Solution

A naive algorithm just checks all possible $(n - 1)!$ cycles.

This part

- Use dynamic programming to solve TSP in $O(n^2 \cdot 2^n)$

Brute Force Solution

A naive algorithm just checks all possible $(n - 1)!$ cycles.

This part

- Use dynamic programming to solve TSP in $O(n^2 \cdot 2^n)$
- The running time is exponential, but is much better than $(n - 1)!$.

Outline

- 1 3-Satisfiability
 - Backtracking
 - Local Search
- 2 Traveling Salesman Problem
 - Dynamic Programming
 - Branch-and-bound

Dynamic Programming

- We are going to use dynamic programming: instead of solving one problem we will solve a collection of (overlapping) subproblems

Dynamic Programming

- We are going to use dynamic programming: instead of solving one problem we will solve a collection of (overlapping) subproblems
- A subproblem refers to a partial solution

Dynamic Programming

- We are going to use dynamic programming: instead of solving one problem we will solve a collection of (overlapping) subproblems
- A subproblem refers to a partial solution
- A reasonable partial solution in case of TSP is the initial part of a cycle

Dynamic Programming

- We are going to use dynamic programming: instead of solving one problem we will solve a collection of (overlapping) subproblems
- A subproblem refers to a partial solution
- A reasonable partial solution in case of TSP is the initial part of a cycle
- To continue building a cycle, we need to know the last vertex as well as the set of already visited vertices

Subproblems

- For a subset of vertices $S \subseteq \{1, \dots, n\}$ containing the vertex 1 and a vertex $i \in S$, let $C(S, i)$ be the length of the shortest path that starts at 1, ends at i and visits all vertices from S exactly once

Subproblems

- For a subset of vertices $S \subseteq \{1, \dots, n\}$ containing the vertex 1 and a vertex $i \in S$, let $C(S, i)$ be the length of the shortest path that starts at 1, ends at i and visits all vertices from S exactly once
- $C(\{1\}, 1) = 0$ and $C(S, 1) = +\infty$ when $|S| > 1$

Recurrence Relation

- Consider the second-to-last vertex j on the required shortest path from 1 to i visiting all vertices from S

Recurrence Relation

- Consider the second-to-last vertex j on the required shortest path from 1 to i visiting all vertices from S
- The subpath from 1 to j is the shortest one visiting all vertices from $S - \{i\}$ exactly once

Recurrence Relation

- Consider the second-to-last vertex j on the required shortest path from 1 to i visiting all vertices from S
- The subpath from 1 to j is the shortest one visiting all vertices from $S - \{i\}$ exactly once
- Hence
$$C(S, i) = \min\{C(S - \{i\}, j) + d_{ji}\},$$
where the minimum is over all $j \in S$ such that $j \neq i$

Order of Subproblems

- Need to process all subsets $S \subseteq \{1, \dots, n\}$ in an order that guarantees that when computing the value of $C(S, i)$, the values of $C(S - \{i\}, j)$ have already been computed

Order of Subproblems

- Need to process all subsets $S \subseteq \{1, \dots, n\}$ in an order that guarantees that when computing the value of $C(S, i)$, the values of $C(S - \{i\}, j)$ have already been computed
- For example, we can process subsets in order of increasing size

TSP(G)

$$C(\{1\}, 1) \leftarrow 0$$

TSP(G)

$C(\{1\}, 1) \leftarrow 0$

for s from 2 to n :

 for all $S \subseteq \{1, \dots, n\}$ of size s :

$C(S, 1) \leftarrow +\infty$

TSP(G)

$C(\{1\}, 1) \leftarrow 0$

for s from 2 to n :

 for all $1 \in S \subseteq \{1, \dots, n\}$ of size s :

$C(S, 1) \leftarrow +\infty$

 for all $i \in S, i \neq 1$:

 for all $j \in S, j \neq i$:

$C(S, i) \leftarrow \min\{C(S, i), C(S - \{i\}, j) + d_{ji}\}$

TSP(G)

```
 $C(\{1\}, 1) \leftarrow 0$   
for  $s$  from 2 to  $n$ :  
  for all  $1 \in S \subseteq \{1, \dots, n\}$  of size  $s$ :  
     $C(S, 1) \leftarrow +\infty$   
    for all  $i \in S, i \neq 1$ :  
      for all  $j \in S, j \neq i$ :  
         $C(S, i) \leftarrow \min\{C(S, i), C(S - \{i\}, j) + d_{ji}\}$   
return  $\min_i\{C(\{1, \dots, n\}, i) + d_{i,1}\}$ 
```

Implementation Remark

- How to iterate through all subsets of $\{1, \dots, n\}$?

Implementation Remark

- How to iterate through all subsets of $\{1, \dots, n\}$?
- There is a natural one-to-one correspondence between integers in the range from 0 and $2^n - 1$ and subsets of $\{0, \dots, n - 1\}$:

$$k \leftrightarrow \{i: i\text{-th bit of } k \text{ is } 1\}$$

Example

k	$\text{bin}(k)$	$\{i: i\text{-th bit of } k \text{ is } 1\}$
0	000	\emptyset
1	001	$\{0\}$
2	010	$\{1\}$
3	011	$\{0,1\}$
4	100	$\{2\}$
5	101	$\{0,2\}$
6	110	$\{1,2\}$
7	111	$\{0,1,2\}$

- If k corresponds to S , how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?

- If k corresponds to S , how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?
- For this, we need to flip the j -th bit of k (from 1 to 0)

- If k corresponds to S , how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?
- For this, we need to flip the j -th bit of k (from 1 to 0)
- For this, in turn, we compute a bitwise XOR of k and 2^j (that has 1 only in j -th position)

- If k corresponds to S , how to find out the integer corresponding to $S - \{j\}$ (for $j \in S$)?
- For this, we need to flip the j -th bit of k (from 1 to 0)
- For this, in turn, we compute a bitwise XOR of k and 2^j (that has 1 only in j -th position)
- In C/C++, Java, Python:
 $k \wedge (1 \ll j)$

Outline

- ① 3-Satisfiability
 - Backtracking
 - Local Search
- ② Traveling Salesman Problem
 - Dynamic Programming
 - Branch-and-bound

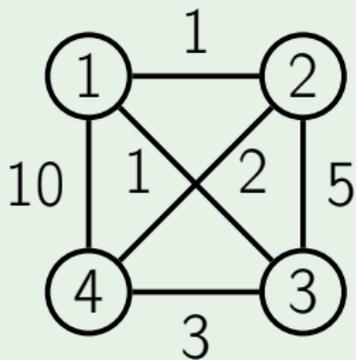
- The branch-and-bound technique can be viewed as a generalization of backtracking for **optimization** problems

- The branch-and-bound technique can be viewed as a generalization of backtracking for **optimization** problems
- We grow a tree of partial solutions

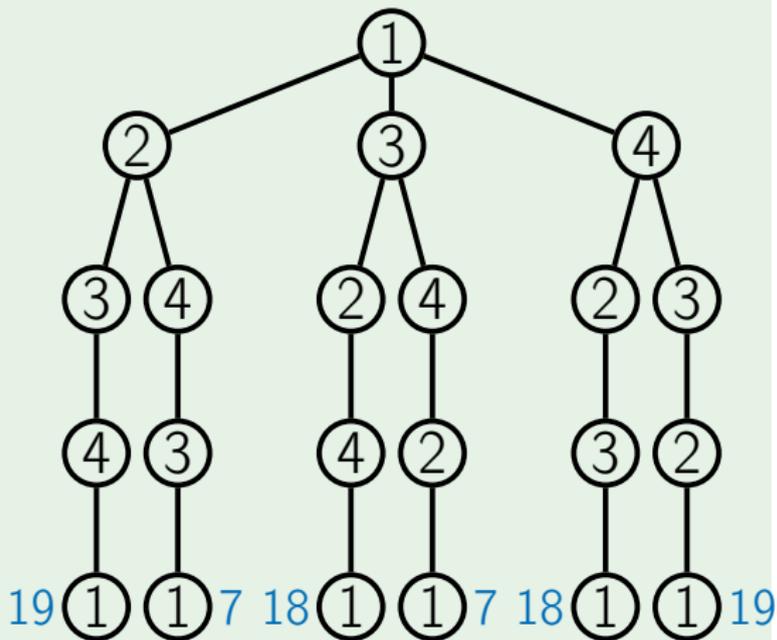
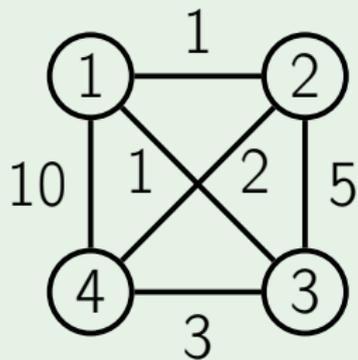
- The branch-and-bound technique can be viewed as a generalization of backtracking for **optimization** problems
- We grow a tree of partial solutions
- At each node of the recursion tree we check whether the current partial solution can be extended to a solution which is better than the best solution found so far

- The branch-and-bound technique can be viewed as a generalization of backtracking for **optimization** problems
- We grow a tree of partial solutions
- At each node of the recursion tree we check whether the current partial solution can be extended to a solution which is better than the best solution found so far
- If not, we don't continue this branch

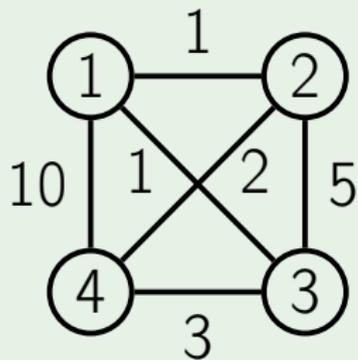
Example: brute force search



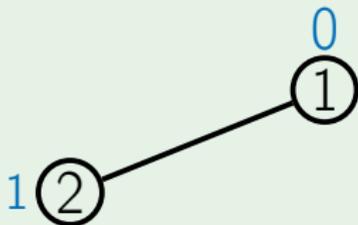
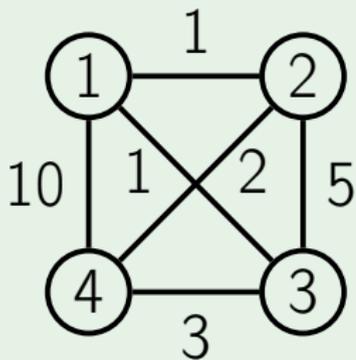
Example: brute force search



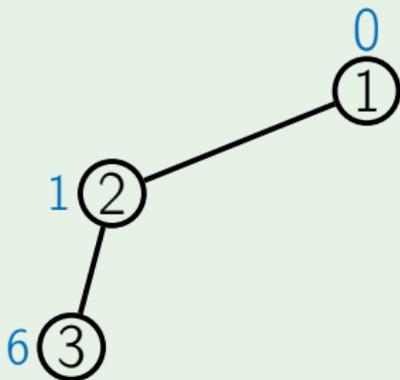
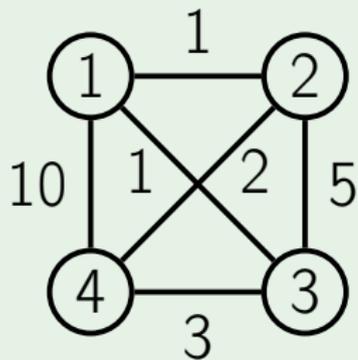
Example: pruned search



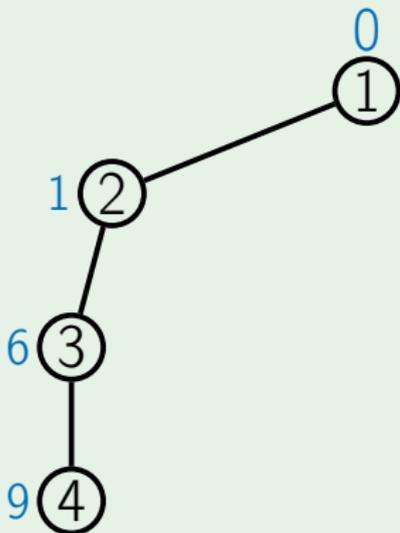
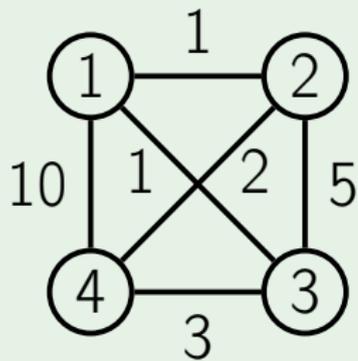
Example: pruned search



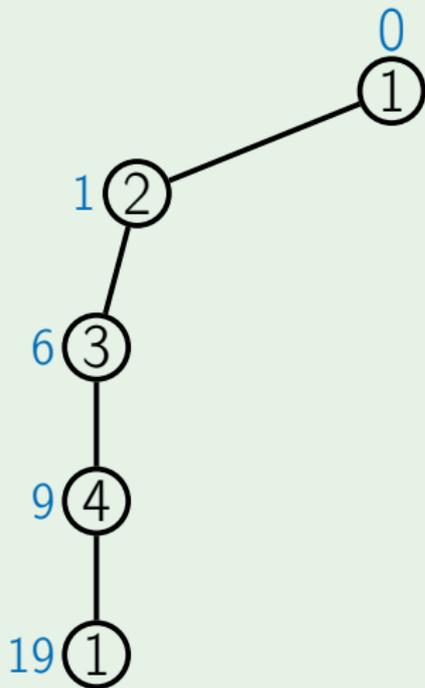
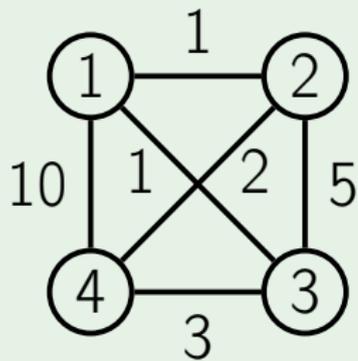
Example: pruned search



Example: pruned search

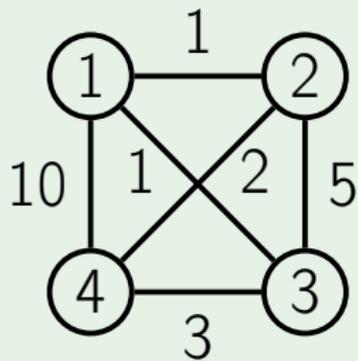


Example: pruned search

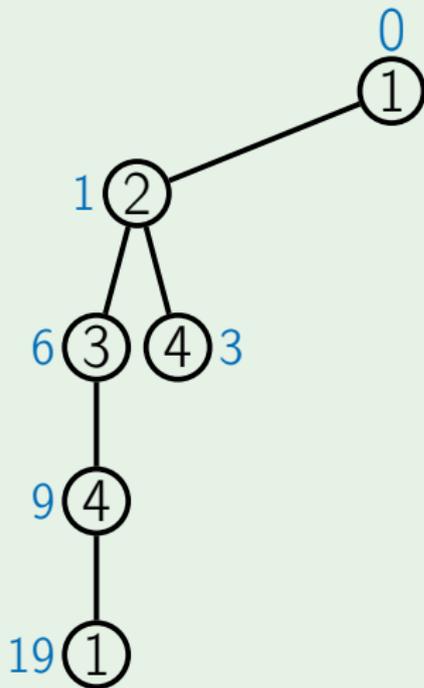


best so far: 19

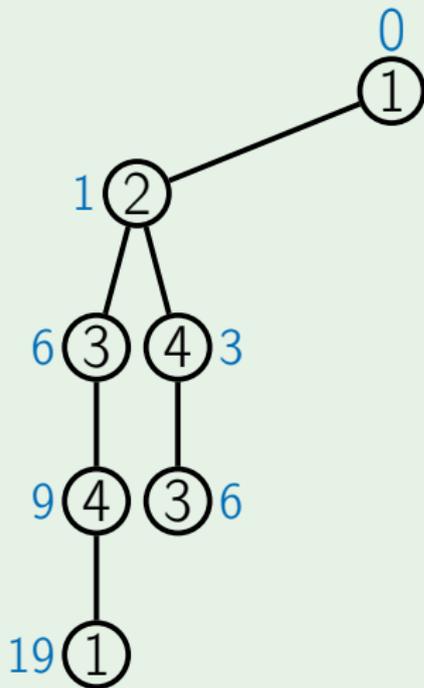
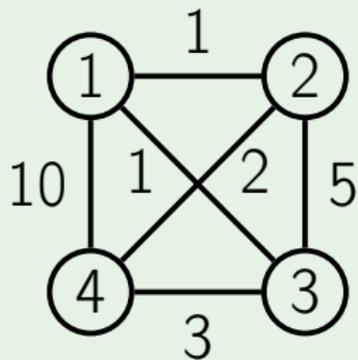
Example: pruned search



best so far: 19

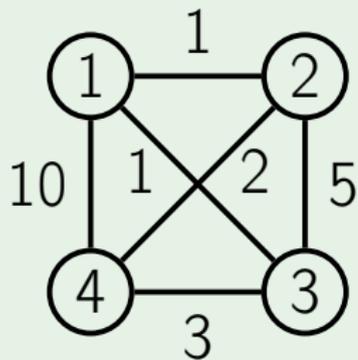


Example: pruned search

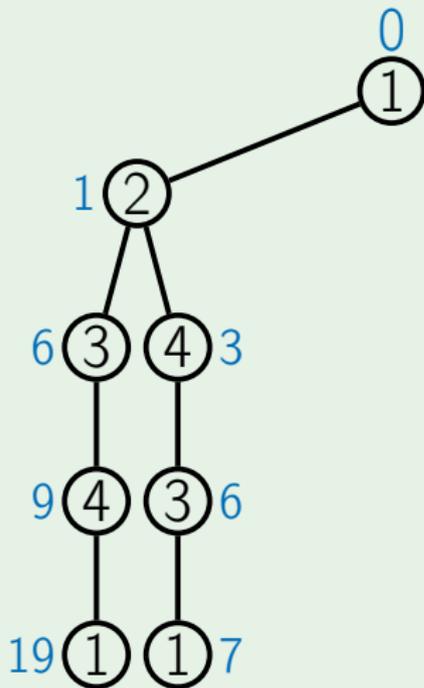


best so far: 19

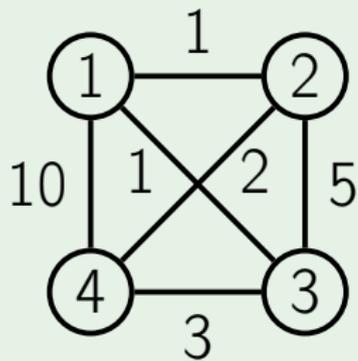
Example: pruned search



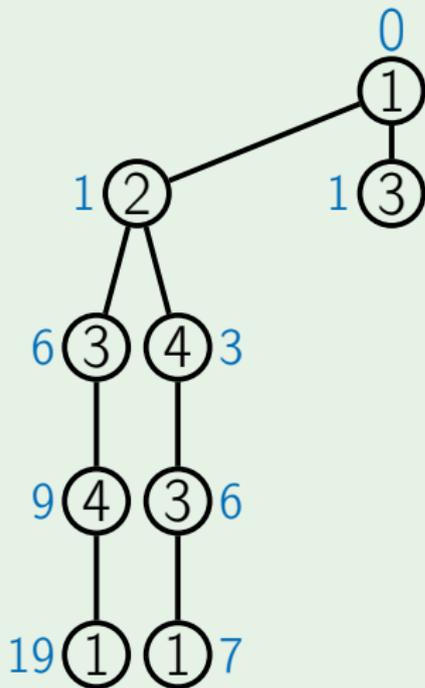
best so far: 7



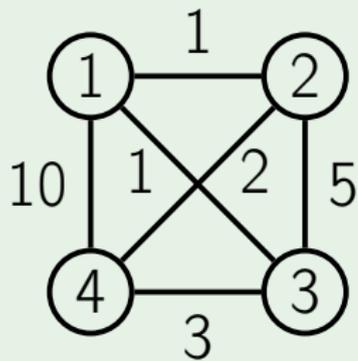
Example: pruned search



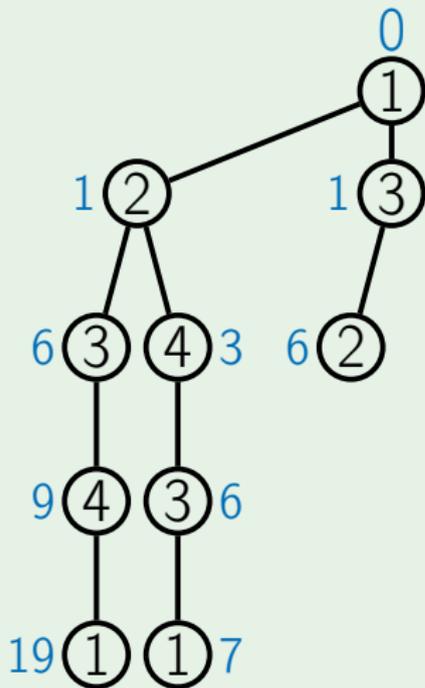
best so far: 7



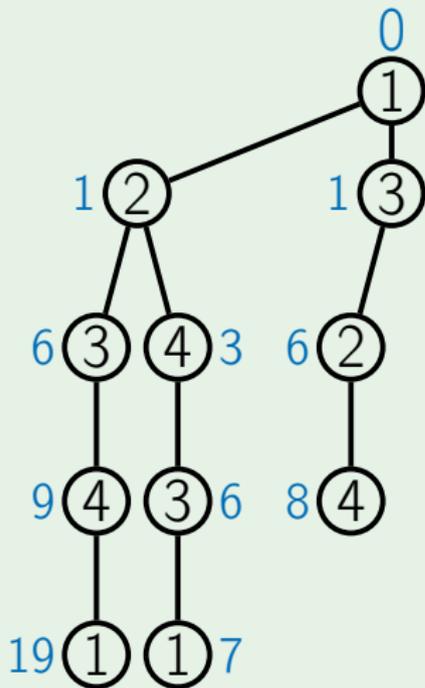
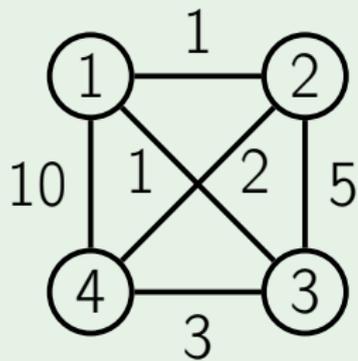
Example: pruned search



best so far: 7

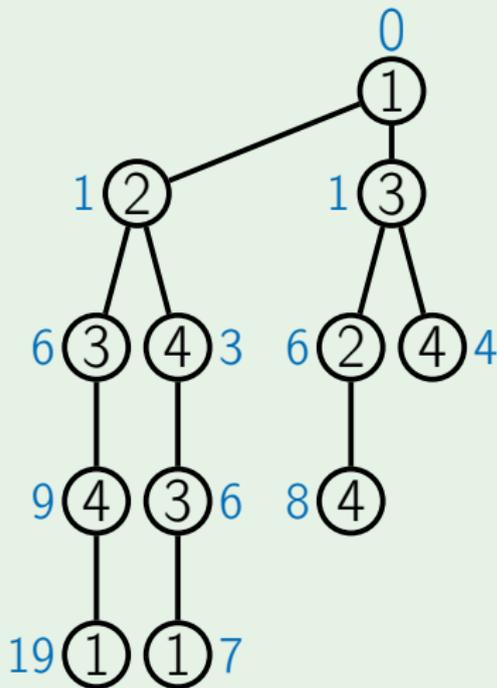
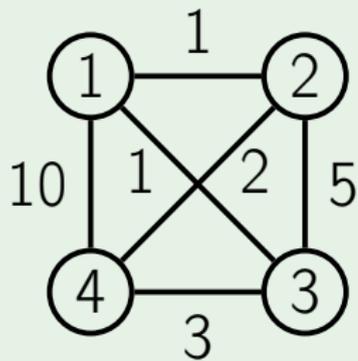


Example: pruned search



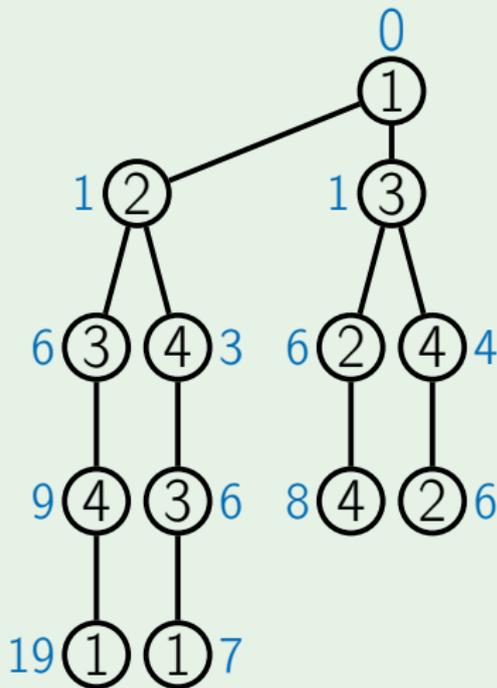
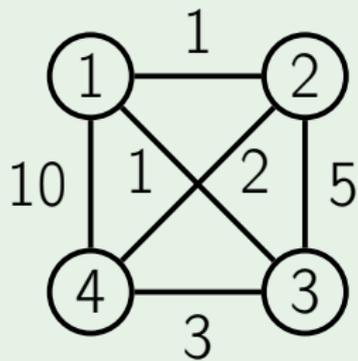
best so far: 7

Example: pruned search



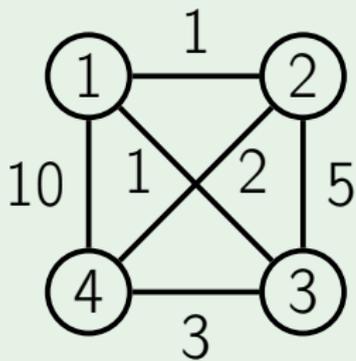
best so far: 7

Example: pruned search

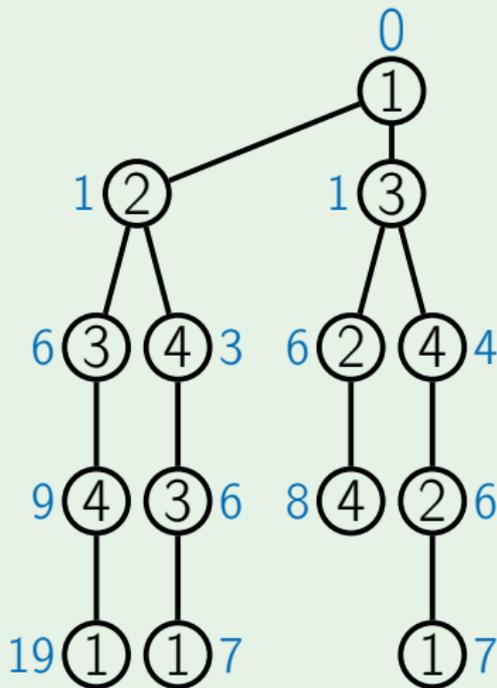


best so far: 7

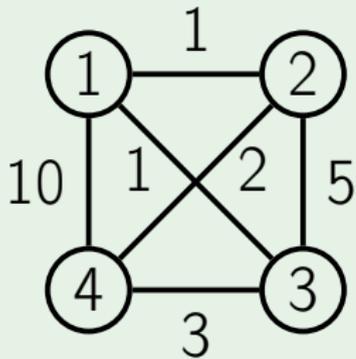
Example: pruned search



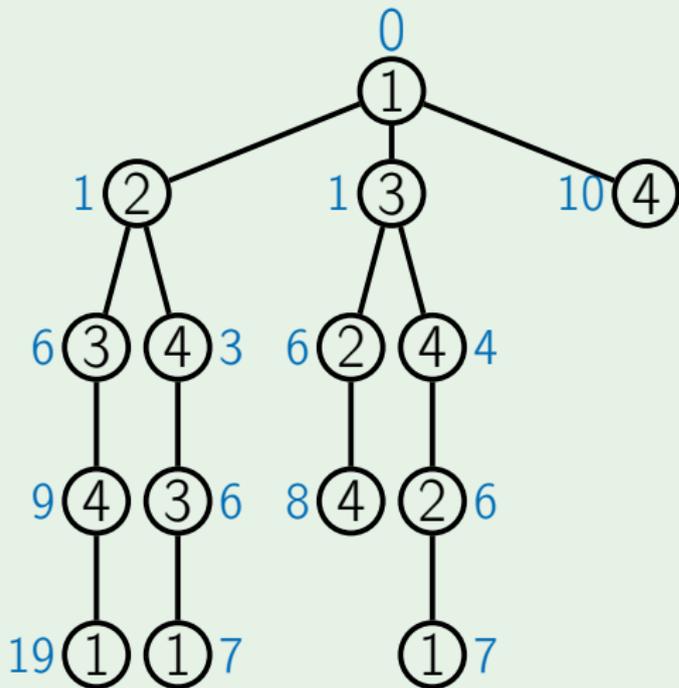
best so far: 7



Example: pruned search



best so far: 7



- We used the simplest possible lower bound: any extension of a path has length at least the length of the path

- We used the simplest possible lower bound: any extension of a path has length at least the length of the path
- Modern TSP-solvers use smarter lower bounds to solve instances with thousands of vertices

Example: lower bounds (still simple)

The length of an optimal TSP cycle is at least

- $\frac{1}{2} \sum_{v \in V} (\text{two min length edges adjacent to } v)$

Example: lower bounds (still simple)

The length of an optimal TSP cycle is at least

- $\frac{1}{2} \sum_{v \in V} (\text{two min length edges adjacent to } v)$
- the length of a minimum spanning tree

Next time

Approximation algorithms: polynomial algorithms that find a solution that is not much worse than an optimal solution